CHAPTER 11

Filtering the Results of Searches

Your application will often need to process only a subset of the data in the database tables. In these cases, you can create a Hibernate *filter* to eliminate the unwanted data. Filters provide a way for your application to limit the results of a query to data that passes the filter's criteria. Filters are not a new concept—you can achieve much the same effect using SQL database views—but Hibernate offers a centralized management system for them.

Unlike database views, Hibernate filters can be enabled or disabled during a Hibernate session. In addition, Hibernate filters can be parameterized, which is particularly useful when you are building applications on top of Hibernate that use security roles or personalization.

When to Use Filters

As an example, consider a web application that manages user profiles. Currently, your application presents a list of all users through a single web interface, but you receive a change request from your end user to manage active users and expired users separately. For this example, assume that the status is stored as a column on the user table.

One way to solve this problem is to rewrite every HQL `SELECT` query in your application, adding a `WHERE` clause that restricts the result by the user's status. Depending on how you built your application, this could be an easy undertaking or it could be complex, but you still end up modifying code that you have already tested thoroughly, potentially changing it in many different places.

With Hibernate 3, you can create a filter restriction for the user status. When your end user selects the user type (active or expired), your application activates the user status filter (with the proper status) for the end user's Hibernate session. Now, any `SELECT` queries will return the correct subset of results, and the relevant code for the user status is limited to two locations: the Hibernate session and the user status filter.

The advantage of using Hibernate filters is that you can programmatically turn filters on or off in your application code, and your filters are defined in your Hibernate mapping documents for easy maintainability. The major disadvantage of filters is that you cannot create new filters at run time. Instead, any filters your application requires need to be specified in the proper Hibernate mapping document. Although this may sound somewhat limiting, the fact that filters can be parameterized makes them pretty flexible. For our user status filter example, only one filter would need to be defined in the mapping document (albeit in two parts). That filter would specify that the status column must match a named parameter. You would not need to define the possible values of the status column in the Hibernate mapping document—the application can specify those parameters at run time.

Although it is certainly possible to write applications with Hibernate that do not use filters, we find them to be an excellent solution to certain types of problems—notably security and personalization.

Defining Filters

Your first step is to define filters in your application's Hibernate mapping documents, using the `<filter-def>` XML element. These filter definitions must contain the name of the filter and the names and types of any filter parameters. Specify filter parameters with the `<filter-param>` XML element. Filter parameters are similar to named parameters for HQL queries. Both require a `:` before the parameter name. Here is an excerpt from a mapping document with a filter called `latePaymentFilter` defined:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping
   PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class ...

  </class>
  <filter-def name="latePaymentFilter">
    <filter-param name="dueDate" type="date"/>
  </filter-def>
</hibernate-mapping>
```

Once you have created the filter definitions, you need to attach the filters to class or collection mapping elements. You can attach a single filter to more than one class or collection. To do this, you add a `<filter>` XML element to each class and/or collection. The `<filter>` XML element has two attributes: `name` and `condition`. The name references a filter definition (for instance: `latePaymentFilter`). The condition represents a `WHERE` clause in HQL. Here's an example:

```
  <class ...
    <filter name="latePaymentFilter" condition=":dueDate = paymentDate"/>
  </class>
```

Each `<filter>` XML element must correspond to a `<filter-def>` element. You may have more than one filter for each filter definition, and each class can have more than one filter. This is a little confusing—the extra level of abstraction allows you to define all the filter parameters in one place and then refer to them in the individual filter conditions.

Using Filters in Your Application

Your application programmatically determines which filters to activate or deactivate for a given Hibernate session. Each session can have a different set of filters with different parameter values. By default, sessions do not have any active filters—you must explicitly enable filters programmatically for each session. The `Session` interface contains several methods for working with filters, as follows:

- `public Filter enableFilter(String filterName)`
- `public Filter getEnabledFilter(String filterName)`
- `public void disableFilter(String filterName)`

These are pretty self-explanatory—the `enableFilter(String filterName)` method activates the specified filter, the `disableFilter(String filterName)` method deactivates the method, and if you have already activated a named filter, `getEnabledFilter(String filterName)` retrieves that filter.

The `org.hibernate.Filter` interface has six methods. You are unlikely to use `validate()`; Hibernate uses that method when it processes the filters. The other five methods are as follows:

- `public Filter setParameter(String name, Object value)`
- `public Filter setParameterList(String name, Collection values)`
- `public Filter setParameterList(String name, Object[] values)`
- `public String getName()`
- `public FilterDefinition getFilterDefinition()`

The `setParameter()` method is the most useful. You can substitute any Java object for the parameter, although its type should match the type you specified for the parameter when you defined the filter. The two `setParameterList()` methods are useful for using `IN` clauses in your filters. If you want to use `BETWEEN` clauses, use two different filter parameters with different names. Finally, the `getFilterDefinition()` method allows you to retrieve a `FilterDefinition` object representing the filter metadata (its name, its parameters' names, and the parameter types).

Once you have enabled a particular filter on the session, you do not have to do anything else to your application to take advantage of filters, as we demonstrate in the following example.

A Basic Filtering Example

Because filters are very straightforward, a basic example allows us to demonstrate most of the filter functionality, including activating filters and defining filters in mapping documents.

In the following Hibernate XML mapping document (`User.hbm.xml`), we created a filter definition called `activatedFilter`. The parameters for the filter must be specified with `<filter-param>` XML elements (as shown in Listing 11-1), which use the `<activatedParam>` XML element. You need to specify a type for the filter parameter so that Hibernate knows how to map values to parameters. Once you have defined your filter, you need to attach the filter definition to a class. At the end of our `User` class definition, we specify that it uses a filter named `activatedFilter`. We then need to set a condition corresponding to an HQL `WHERE` clause for the attached filter. In our case, we used `:activatedParam = activated`, where `:activatedParam` is the named parameter specified on the filter definition, and `activated` is the column name from the user table. You should ensure that the named parameter goes on the left-hand side so that Hibernate's generated SQL doesn't interfere with any joins.

Listing 11-1. *Hibernate XML Mapping for* `User`

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping
   PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="com.hibernatebook.filters.User">
    <id name="id" type="int">
      <generator class="native"/>
    </id>

    <property name="username" type="string" length="32"/>
    <property name="activated" type="boolean"/>
    <filter name="activatedFilter" condition=":activatedParam = activated"/>
  </class>
  <filter-def name="activatedFilter">
    <filter-param name="activatedParam" type="boolean"/>
  </filter-def>
</hibernate-mapping>
```

With the filter definition created and attached to a class with a suitable condition, we need to activate the filter. The next class, `SimpleFilterExample`, inserts several user records into the database, and then immediately displays them to the screen. The class uses a very simple HQL query (`from User`) to obtain the result set from Hibernate. The `displayUsers()` method writes the usernames and activation status out to the console. Before you have enabled any filters on the database, this method will return the full list of users. Once you have enabled the first filter (`activatedFilter`) to show only activated users, call the same `displayUsers()` method—the results of the query are the same as if you had added a `WHERE` clause containing an `"activated=true"` clause. You can just as easily change the filter's parameter value to show inactive users, as shown in Listing 11-2.

Listing 11-2. *Invoking Filters from Code*

```
package com.hibernatebook.filters;

import java.util.Iterator;

import org.hibernate.Filter;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

import org.hibernate.cfg.Configuration;

public class SimpleFilterExample {
    public static void main (String args[]) {
        SessionFactory factory =
           new Configuration().configure().buildSessionFactory();
        Session session = factory.openSession();

        //insert the users
        insertUser("ray",true,session);
        insertUser("jason",true,session);
        insertUser("beth",false,session);
        insertUser("judy",false,session);
        insertUser("rob",false,session);

        //Show all users
        System.out.println("===ALL USERS===");
        displayUsers(session);

        //Show activated users
        Filter filter = session.enableFilter("activatedFilter");
        filter.setParameter("activatedParam",new Boolean(true));
        System.out.println("===ACTIVATED USERS===");
        displayUsers(session);

        //Show nonactivated users
        filter.setParameter("activatedParam",new Boolean(false));
        System.out.println("===NON-ACTIVATED USERS===");
        displayUsers(session);
        session.close();
    }

    public static void displayUsers(Session session) {
        session.beginTransaction();
        Query query = session.createQuery("from User");
        Iterator results = query.iterate();
        while (results.hasNext())
        {
            User user = (User) results.next();
            System.out.print(user.getUsername() + " is ");
            if (user.isActivated())
            {
                System.out.println("activated.");
            }
            else
            {
                System.out.println("not activated.");
            }
        }

        session.getTransaction().commit();
    }

    public static void insertUser(String name, boolean activated, Session session) {
        session.beginTransaction();

        User user = new User();
        user.setUsername(name);
        user.setActivated(activated);
        session.save(user);

        session.getTransaction().commit();
    }
}
```

The output of `SimpleFilterExample` is as follows:

```
===ALL USERS===
ray is activated.
jason is activated.
beth is not activated.
judy is not activated.
rob is not activated.
```

```
===ACTIVATED USERS===
ray is activated.
jason is activated.
===NON-ACTIVATED USERS===
beth is not activated.
judy is not activated.
rob is not activated.
```

Listing 11-3 gives the `User` class used for this chapter's examples. The only fields it contains are `id`, `username`, and `activated`.

Listing 11-3. *The Source Code for the* `User` *Class*

```
package com.hibernatebook.filters;

public class User {
    private int id;
    private String username;
    private boolean activated;


    public boolean isActivated() {
        return activated;
    }

    public void setActivated(boolean activated) {
        this.activated = activated;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

Because filters do not use any database-specific functionality beyond the Hibernate configuration, you should not encounter any difficulty running this example on databases other than HSQLDB. The Hibernate configuration file defines the database configuration and connection information, along with the XML mapping document for the `User` class (see Listing 11-4).

Listing 11-4. *The Hibernate XML Configuration File for the Example*

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">
           org.hsqldb.jdbcDriver
        </property>
        <property name="hibernate.connection.url">
           jdbc:hsqldb:file:filterdb;SHUTDOWN=true
        </property>
        <property name="hibernate.connection.username">sa</property>
        <property name="hibernate.connection.password"></property>
        <property name="hibernate.connection.pool_size">0</property>
        <property name="dialect">
           org.hibernate.dialect.HSQLDialect
        </property>

        <!-- Mapping files -->
        <mapping resource="com/hibernatebook/filters/User.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

The source code for this chapter includes the schema we used for the HSQL database to create the table for the `filterdb` database.

Summary

Filters are a useful way to separate some database concerns from the rest of your code. A set of filters can cut back on the complexity of the HQL queries used in the rest of your application, at the expense of some runtime flexibility. Instead of using views (which must be created at the database level), your applications can take advantage of dynamic filters that can be activated as and when they are required.